APPENDIX A

Introduction

Appendix A will be understood and appreciated from the following detailed description, taken in conjunction with the drawings in which:

Fig. 43 is a simplified block diagram showing relationships between data sources and a processing module constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 44 is a simplified block diagram showing relationships between data sources and a pre-emptive processing module constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 45 is a simplified block diagram showing user defined tasks of a processing module, constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 46 is a simplified block diagram illustrating a composite processing module constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 47 is a simplified block diagram illustrating input channels into a pre-emptive processing module and a cooperative processing module, constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 48 is a simplified block diagram illustrating output channels from a pre-emptive processing module and a cooperative processing module, constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 49 is a simplified block diagram illustrating a registration processing module constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 50 is a simplified block diagram illustrating a root process manager, constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 51 is a simplified block diagram illustrating an execution graph scenario constructed and operative in accordance with a preferred embodiment of the present invention;

Fig. 52 is a simplified block diagram illustrating a root directory tree from which the SIP begins, constructed and operative in accordance with a preferred embodiment of the present invention; and

Fig. 53 is a simplified flow chart illustrating SIP states and the commands that effect them, constructed and operative in accordance with a preferred embodiment of the present invention;

Presented is an architecture of the kernel of the SIP (software image processing) that is capable of running efficiently on SMP (symmetric multi-processing) machines (e.g., a multiprocessor Intel based machines), and on a single processor machine. The implementations of object oriented techniques to implement multiprocessing in a platform independent way is discussed.

The goal of the SIP kernel is to allow easy and efficient processing of different scan scenarios. A scan scenario usually involves connection of the scanner's hardware, reading incoming information about the scanned board and processing this information in different ways. For example, in a learn scan scenario we create a reference of the board from the incoming data, and while in the inspect scenario we compare this reference to the actual data coming from the scanner. If the inspected board is too different from the reference board, we decide that the board is defective. The scan scenario may involve only off-line computations, without connecting to any scanner. We may look at the SIP as a tiny operating system that is responsible for carrying out efficiently many different scan scenarios.

The SIP schedule is a set of processing modules. Each processing module consumes zero or more data sources and produces zero or more data sources. A data source is any data structure shared by more than one processing module.

In this appendix changes to two basic entities in the SIP - the task/channel and the data source - are discussed. The separate task and channel objects are preferably joined as a new "meta-task" object which can function both as a task and as a channel. The channel capabilities of this new object are provided by a new kind of data source which encapsulates the access to data via hardware - the Stream_data_source. A preferred modification to the data sources is the capability to allocate a data source in shared

memory, thus allowing "meta-tasks" executing in more than one process to share data efficiently and transparently.

Definition of an abstract processing module

This abstract base class represents the most basic computational element that can be defined within the SIP. A Processing module preferably comprises a manager which is a method/program that initiates and manages it.

Reference is now made to Figure 43 which is useful in understanding a preferred embodiment of the present invention.

A processing_module may have up to three kinds of "relationships" with data sources:

Consumer of a data source (reader) - reads and processes data from these data source(s).

Asynch Consumer of a data source (reader) - the processing module registers to be notified by the data source whenever new data is available for reading and processing.

Producer of a data source (writer) - writes produced data to produced data sources.

A processing module typically has at least one consumed data source to be correctly configured. If a processing module does not consume any data source, it will not get scheduled to process.

The basic Processing Module performs some sort of processing on consumed data. Following are several examples of concrete derived classes (which are discussed in greater detail in the following sections):

Some sort of computation performed on input DS (data source) with results written to the output DS.

A parsing operation on input data, with parsed data written to selected data source.

A formatting operation on input data, to write it to an output stream data source.

The Processing Module is defined in the SIP configuration file.

Another basic trait of the abstract base class is now defined:

Definition of a pre-emptive processing module versus cooperative.

Reference is now made to Figure 44 which is useful in understanding a preferred embodiment of the present invention.

A pre-emptive processing module (denoted by the broken lines in Figure 44) exists within an independent operating system process/ thread. As such, it is scheduled to be run by the operating system according to system level events. A cooperative Processing_Module is scheduled by it's manager, usually a higher level composite processing module scheduler.

All data sources referenced by a pre-emptive processing module are preferably placed in shared memory, denoted by the broken lines. This is preferably done automatically by the configuration file loader.

Indication of data available for processing/ end of data etc. is typically signalled in the following manner by the updated data source:

For pre-emptive processing modules, communication will be through pipes which are to established by every pre-emptive processing module.

For a cooperative processing module active signalling is not typically needed. The update of more data ready to process.

Every pre-emptive processing module processing module has a unique ID which is its operating system process ID. To this ID the file descriptor associated with the pre-emptive process pipe is attached. Data sources which are dependent on streams external to the SIP will also provide an external file descriptor to which the owning pre-emptive processing module will be attached.

The attachment of file descriptors to pre-emptive processing modules is performed after the load configuration process. This may be a portability issue because there are platforms which allow only one un-named pipe per processes.

The consumer registration process preferably works as follows:

The consuming processing module calls Register_Consumer() of the data source, providing the data source with its pipe object.

The data source method Register_Consumer() will return a pipe object on which the consuming processing module is to sleep. If the data source is an internal SIP data source, it will return the same pipe object with which it was called, indicating that it will send a wake up call to the registered consumer's wake up pipe. If the data source is a representation of an external data stream, it will return a different "pipe object" to which the external file descriptor is attached.

The Register_Consumer() method accepts a grain size parameter, which determines how often data available notifications will be sent. If a gain size of zero is selected, than notifications will get sent only for the following events: The data source has reached end_of_data status (end of scan), or by an explicit notification call made by a producer.

Data sources produced by a processing module now fall into two categories - shared and local. Writing data to a local data source is effected without making changes. It gets more interesting when writing data into a shared data source: if a pointer to data allocated by the task on a heap is provided, then it needs to be ensured that data is allocated on the shared memory heap, otherwise the pointer will be invalid in the context of a different pre-emptive processing module.

When a data source determines that it has sufficient data for filling a data grain as registered by one of it's consumers, it sends an alert to the consumer. The alert does nothing if the consumer is a local task, and sends a notice to the consumer pre-emptive processing module's pipe.

A data source's producer may call a flush() method of the data source which will cause it to wake up all of its consumers. This will occur either at end of scan or when the processing of a big chunk of data has been completed.

Concrete Derived Classes - a sample taxonomy

Reference is now made to Figure 45 which is useful in understanding a preferred embodiment of the present invention.

Following are several examples of processing modules:

The user defined task may be of several types:

Composite processing Module - Execution graph - Some sort of computation performed on input data with results written to the output, and two types are typically available - cooperative and pre-emptive.

Input Channel - A parsing operation on input data, with parsed data written to selected data source. The input data will come from an input stream data source, which will be attached to either a file, TCP/IP or a DMA (Direct Memory Access) channel. Writing to a selected data source will allow step and repeat processing in the future, with data from different repeats dispatched to different output data sources which each serve as input to independent execution graphs.

Output Channel - A formatting operation on input data, in preparation to writing the input data to an output stream data source. The output stream may end up in a file or in a remote computer via a TCP/IP connection.

Registration Transform Producing Task - A computational task which consumes input scan data source non-Asynchronously and writes both an updated transform to a scan line transform data source and excess-missing reports to an excess-missing data source.

A detailed description of these processing module derived classes follows.

Definition of a composite processing module

Reference is now made to Figure 46 which is useful in understanding a preferred embodiment of the present invention.

A composite processing module typically consists of several processing modules which are mostly interconnected between each other. All data sources which are used locally only are placed in local scope and their names are not available globally.

Only data sources which are external to the local processing modules are connected to the external scope processing modules. The connection is maintained through the composite processing module which schedules the internal processing modules when notified by the external data sources.

Because a pre-emptive processing module cannot be scheduled by a SIP scheduler object, a composite processing module may not have pre-emptive processing modules in it's execution graph.

Every processing module may be a composite (and thus have many processing modules "embedded" in it). A preferred limitation is that all "embedded" processing modules of a pre-emptive composite processing module are typically cooperative so that they may be managed by it.

This limitation defines a constrained general processing module structure: all of the first level processing modules are preferably pre-emptive, and all these embedded in a first level composite pre-emptive processing module are preferably cooperative.

An input channel

Reference is now made to Figure 47 which is useful in understanding a preferred embodiment of the present invention.

An input channel from DMA is constructed by coupling a local Input Stream data source as the single consumed data source of a processing module parsing task.

When the processing module is pre-emptive, the parsing task is awakened to work only when input data is available in the stream data source, which is local to the context of the input channel process. When the processing module is cooperative it will work continuously on the input stream.

The input channel process sleeps both on the command pipe (for commands from the root SIP process) and on the underlying stream system level File Descriptive.

This design assumes that whenever data is available on the input stream, the channel process will read as much data as possible, parse it and write it to the produced data sources. This design works well when the input stream is based on an external data provider such as either DMA or a TCP/IP stream.

When working with a local file, reading the entire file in may cause the produced data sources to overflow. Obviously, working with a local file typically requires some scheduling beyond that provided by the operating system.

The solution is to place local file input stream data sources within a cooperative composite processing module which will schedule reading of data in well sized chunks. This solution will work well, since a local file may be considered synchronous and typically does not require a separate operating system process to handle access to it.

An output channel

Reference is now made to Figure 48 which is useful in understanding a preferred embodiment of the present invention.

An output channel is constructed by coupling a local Output Stream as the single produced data source of a processing module which formats (or serializes) it's input Asynch shared data sources into the output stream.

The processing module may be either pre-emptive or composite:

A pre-emptive processing module typically is used when the output stream is directed to some "blocking" system device such as a TCP/IP connection, or when output to a file is under very low priority and is to be scheduled only when the system is idle.

When the stream is directed to a regular file, the operating system is preferably able to handle the write requests without blocking the writing process. Therefore, an Ostream data source which writes to a file could be placed within a cooperative composite processing module without significant loss of performance (except that related to the actual write to file).

A Registration Transform Cooperative Processing Module.

Reference is now made to Figure 49, which is useful in understanding a preferred embodiment of the present invention.

The registration task is implemented as a "Registration processing module", which may be implemented as a pre-emptive or cooperative module. This task consumes a scan line data source, and produces a transform scan line data source and an Excess-Missing data source.

When the registration task is cooperative, it is implemented within a composite processing module and is scheduled to process data by it's composite owner's scheduler.

When the registration task is pre-emptive, it sets up a either timer which awakens it at timeout or defines a large data grain for notification and wake up.

Definition of the Root Processing Module (the Process Manager).

Reference is now made to Figure 50, which is useful in understanding a preferred embodiment of the present invention.

The root process has three objects which are activated by a main loop which is entered after the SIP has been configured by the application. Prior to configuration, the only active object is the interface to the application. The SIP typically has at least one of the two processing objects - either a Processing Module Manager and/or a local composite cooperative processing module.

The three objects are:

Interface to the application - an object which handles all communication and parsing of the commands sent to the SIP by the application.

Local single composite cooperative processing module. This module is made available in the root process for the following reasons:

* To allow easy debugging of new execution graph modules by developers. This is by letting a complete execution graph run in one debuggable process.

* For allowing the running of "stand-alone" SIP programs (such as a background learn- update).

* Backwards compatibility during the development phase of multi-processing capabilities.

Processing Module Manager - an object which handles all communication with the stand alone pre-emptive processing modules.

Every pre-emptive processing module sets up one pipe on which it listens for messages from the other processes in the SIP group of processes. The root manager process sets up a similar pipe to listen to acknowledgements/ messages from the managed pre-emptive processing modules.

The following kinds of messages may pass in the system between the processes in the group:

Stop/Status Query commands from the root processing_module manager to the pre-emptive processing modules.

Command acknowledgement and error state/ end of scan reporting back to the root process.

Indication of data gain availability/ end of data/ error in data in a shared data source - the data source notifies all consumers which had registered with it for notification.

The message is typically passed as a stream of characters. The basic message format is:

Field Position [char]	Field Type	Description
1	message character	Message represented by character: commands from root: 's' - stop scan; 'q' - query status (ping); 't' terminate child process; 'r' - reset to initscan condition, 'u' - do uninitscan, 'b' - start scan, 'f' - go to endscan(ok), 'g' - endscan(error) acknowledgement to root: 'k' - child killed; 'x' - child in error mode; 'v' - child running; 'i' - child idle in scan } child in INIT state 'e' - child at end of scan state 'y' - child in OK state. message from other SIP process: 'd' - notification of change of status in a registered consumed data source to pre-emptive consumer.

2:6	sender SPID	The message sender's system process ID in ascii (00000-99999)

The Root process will typically "barrier sync" all child processes so that when a new command is issued they all start out in the same state.

Data Source Modifications

There are two main changes which are to performed on current SIP data source objects.

One change is typically used by selecting Multi-Processing - the capability to allocate a data source is shared memory which is visible in all the SIP group of processes. The data source preferably implements the minimal level of access protection/ mutual exclusion used by the current spec for access to this data source.

The second change is typically used due to the merging of the single process SIP task and channel objects to the "Processing_module" object - the adding of stream data sources.

Also, the helper classes Sip_mem, chain_of_blocks and reservoir which manage memory (Such as a memory model for a DS_Array type of data source) is preferably shared memory compatible. This compatibility means:

1. ALL allocations are typically made in the shared memory heap.

2. Mutex / Reader-Writer access locks - to be used sparingly as possible, with a single writer- multiple reader atomic write capability allowing no mutex's to be utilized.

Data sources in shared memory

The allocation of data sources in shared memory is typically directed by the SIP class configuration file reader, which selects the correct allocation mode according to the context.

Data sources placed in the top-level configuration file are typically allocated in shared memory. When loading configuration files for subsequent composite Processing_module (s) the data sources are preferably built locally..

A data source is typically created in shared memory by the sip_factory by specifying an optional second parameter to its *clone* method as "true". The parameter defaults to false for default objects constructed on the local heap.

The writer of a data source preferably provides an extra implementation of a shared memory creator method;

A task which interacts with a shared data source (and provides the data source with pointers to data which the task allocates) typically checks if any of the data sources are in shared memory, and allocates its data accordingly.

A shared memory data source preferably allocates all its dynamic storage on the shared memory. This does not necessarily mean that a different implementation is used for a local heap/ shared heap data source. Some alternative options include:

1. Turn current data source into an abstract class, and provide two derived classes - one for shared memory and one for local. Define two not valid methods in the derived classes to return errors. This forces the execution graph creator to specify a different kind of class.

2. Allocate data source to dynamic allocations in shared memory, which then allows the user to safely use the interface described above to select shared / local data sources. This using the shared heap may be slower than using the local heap, due to the mutex lock used by the shared new operator to protect its own internal data.

Following from the previous solution, foregoing direct memory allocations and using private memory pools which manage the shared/ non shared issues is probably the most robust solution.

A third approach, which seems more robust and user friendly, is selected for the initial implementation.

It is appreciated that shared data sources may get pointers to data allocated by the producing task. Any data which is passed this way is preferably allocated in shared memory by the task. Tasks are preferably checked to see if they are shared memory compatible. An example of an item which is affected by this is the memory_model which allocates and handles data for ds_array class objects.

Stream data sources

A stream data source is an encapsulation of a SIP i/o utility object, which provides it with an interface compatible with a generic data source. This interface allows the coupling of a Stream_data_source object to a processing module and the creation of an I/O (input/output) channel.

The basic abstract class would be Stream_data_source which is inherited from data_source. This basic class provides a generic I/O wrapper for io_util. Concrete derived classes include:

IStream_data_source - An encapsulation for an input from FILE, TCP/IP or (future) DMA.

OStream_data_source - An encapsulation for output to a file or TCP/IP.

Step and Repeat Considerations

A preferred method for handling step and repeat situations is to define the small reference data base, create several execution graphs (the number being defined by how many repeats can be handled in parallel) and to have the input channel send the incoming scan data to the execution graph corresponding to a certain repeat.

Reference is now made to Figure 51, which is useful in understanding a preferred embodiment of the present invention.

The execution graph scenario typically appears as in Figure 51 with at least five pre- emptive processing modules. If a decision is made to split up the per-repeat execution graph, then there typically will be eight or more pre-emptive processing modules.

The logic typically used to handle the step and repeats are typically incorporated into *three* modules in the system:

The root process is preferably aware of the current scan data repeat number, and which execution graphs are busy on which repeats. The root process manages the execution graph priorities so that the oldest scan still processed will be of the highest system priority.

The input channel pre-emptive processing module will know how to detect a change in repeat number from the incoming scan data, and switch the output of parsed scan data into the appropriate data sources. It typically selects the data sources according to the table of free execution graphs maintained by the root process.

The output channel pre-emptive processing module preferably knows which execution graph is processing which repeat, in order to buffer and format its output of data to the application.

End of scan is typically handled in a slightly more complex way: An end of scan condition reported by an execution graph will typically be interpreted as "end of repeat processing" by the root process, which then sends the execution graph a "InitRepeat" message which will reset it for handling the next repeat. The SIP typically switches to end of strip-scan condition when the input channel pre-emptive processing module signals "end of scan" and after all the execution graphs and the output channel pre-emptive processing module have signalled "end of scan".

In a more complex scenario, several kinds of execution graphs may be supported concurrently: e.g., in a learn - scan -scan ..... scenario. The above mentioned architecture scales well to this preferred requirement: The table of free execution graphs is preferably extended to hold the type of execution graph, so that any number of classes of execution graphs can be handled by the root process.

Detailed Class Definitions:

Documentation is typically generated on-line automatically for various classes (Fig. 53).